SHARP APL
OCTOBER-DECEMBER 1975

Technical Supplement

by Clement Kent

⎕FD AND FUNCTIONS OF FILES

In the last supplement there were a number of examples of the use of 3 *⎕FD* to define and execute local functions. A utility function called *XEQ* was used to execute an *APL* expression which, in the form of a character vector, was the right argument *CODE* of *XEQ*. The following is a slightly more general version of *XEQ*:

```
      ∇ R←XEQ CODE;FOO
[1]    →(2=ρR←ρCODE)ρMAT
[2]    CODE←'∇FOO',CR,'[1] ',CODE,CR,' ∇' ⋄ →DEF
[3]   MAT: CODE←(R[2]↑'FOO'),[1] CODE
[4]   DEF: →(0≐1↑0ρR←3 ⎕FD CODE)ρ0
[5]    FOO
      ∇
```

Note that this *XEQ* is slightly different from the one used last issue, since it returns the diagnostic result *R* of 3 *⎕FD*. This result can be used outside *XEQ* to check that the *CODE* was well formed.

The *CODE* argument can now be either a vector corresponding to one line of *APL*, or a matrix with one row per line. Thus, to execute several expressions at once, we need only put them in a vector separated by diamonds, or in a matrix with one expression per row.

One use of this more general *XEQ* is in data base management, specifically in the definition of *PUT* and *GET* functions.

```
       ∇ GET ACCOUNTS
[1]    XEQ '⍞A⍞,LI10,⍞←⎕READ ⍞,3I10' ⎕FMT (ACCOUNTS; FCOMP ACCOUNTS)
       ∇

       ∇ PUT ACCOUNTS
[1]    XEQ '⍞(6 ⎕WS A⍞,LI10,⍞) ⎕REPLACE ⍞,3I10' ⎕FMT (ACCOUNTS; FCOMP
                                                              ACCOUNTS)

       ∇
```

Here *FCOMP* is a user-defined function which takes a vector of account codes and returns a matrix of the file and component numbers in which the data for those accounts can be found. *GET* reads the data into variables with standard names; the data for account 153 would be in the variable named *A*153. Conversely, *PUT* will put the data in these variables into the right places on the file. This data base access method allows users to write very simple programs without worrying about files. An example:

```
       ∇ UPDATE
[1]    GET 153 200 310
[2]    A200←A200×A153
[3]    A310←A310+A200
[4]    A450←A310-0.15×A310
[5]    PUT 200 310 450
       ∇
```

Here accounts 153, 200 and 310 are read in, 200 and 310 are changed, a new account number, 450, is calculated, and 200, 310 and 450 are replaced on the file. A similar procedure is used in the *AIDS* system, where the file structure for *AIDS* time series is completely transparent to the user.

XEQ may be used in program-controlled execution of commands entered on the terminal - very useful in interactive teaching systems. It could be used in a flexible customer billing system, in which all information regarding discounts and billing exceptions for customers are encoded as *APL* expressions in a file. When the billing run is done, any non-standard charges can be accomodated in the general billing program by reading in and executing the special expressions for each customer. This approach is being used in the Sharp customer billing system.

One major reason for placing *APL* statements on a file is to save space - short programs of a general nature can reside in the workspace, while the exceptions and special cases can be put on a file. This can produce an extremely flexible system if properly done. Each user - salesman, programmer, executive - can program just those cases or problems of particular interest. If all program code is in a central file rather than being scattered through several workspaces, it is easier to document and modify, and therefore easier to maintain.

SYSTEMS USING FUNCTIONS ON FILES

The code for a function can be kept in a file component in 1 or 2 ⎕*FD* format. It can be read into the workspace when needed and turned into a local or global function via 3 ⎕*FD*. This allows the user to build a system in which overcrowded workspaces are replaced by a single workspace containing a central function which can get subfunctions from the files and execute them.

Two systems using filed functions are discussed. First, a hybrid system designed for a particular application, then a general system with many possible uses.

THE *MEMORE* SYSTEM is a Delphi Conferencing and text handling package developed for General Conferencing Systems. It is oriented towards users who are unfamiliar with computers; and thus must combine the ability to execute a large list of user commands on request and a considerable amount of error checking and input validation. The programs required to do this fill several workspaces, but they must all be available to the user on demand. To avoid *WS FULL* problems, most programs are kept in a central file and are read into the workspace when needed. All such programs are local to an *XEQ*-type function, so that they are erased after execution. To avoid excessive file overhead, commonly used utility functions are kept in the workspace. Thus, some commands are done by programs in the workspace and some by filed programs. This hybrid system is a reasonable compromise between the problems of *WS FULL* and the overhead costs of functions on file.

It is possible to transfer the contents of an entire workspace to a file in such a way that the workspace can be reconstituted from the file under program control (unless the workspace contained groups). The *WSTOFILE* function used by the 7 *WSDOC* programs can transfer a workspace to a file in a format suitable for the *WSDOC* program. In the other direction, the first major system capable of running entirely from filed functions was the SPOE Program Maintenance and filing system (*SPM*), developed for Xerox, Inc. by Ian Griggs and Peter Teeson of Sharp's Toronto consulting group. The following description of *SPM* is adapted from Ian's provisional documentation.

THE SPOE PROGRAM MAINTENANCE AND FILING SYSTEM is a general-purpose set of *APL* functions which

1) store and retrieve functions on files using ⎕*FD*
2) produces various listings and reports to facilitate programming and debugging.

SPM may be used for the construction and operation of any collection of *APL* functions that work together as a single system. Complex, large-scale, interactive *APL* systems are made accessible to unskilled users who cannot be expected to knowledgeably or reliably use *APL* system commands. Such systems might include elaborate, on-line inventory systems, financial reporting systems with a great variety of options, systems for highly selective, carefully guided inquiries into a complex data base, and so on.

If a system of functions will fit comfortably into a single workspace, *SPM* is unlikely to prove useful. If a system is used only by an *APL* programmer (for example, a one-shot set of report functions, or a monthly file creation) the careful structuring of a system required by *SPM* may be unnecessary. *AUTOLOAD* is a more economical facility for batch-type systems, where phases of a system are executed independently in a predetermined sequence.

One significant advantage of *SPM* is that in-line comments may be used freely. All comments are automatically deleted when executable code is set up. The system is therefore very easy to maintain.

Principles of Operation

The basic idea of *SPM* is the "local function". With the use of ⎕*FD*, a function may be made local to function *A*, just as a variable may be made local. *B* has, of course, no value inside *A* until it is defined using ⎕*FD*; like a local variable, it exists only while *A* is active and is effectively erased when *A* is exited. Given an appropriate method of automatically defining *B* in the workspace whenever *A* is called, this dynamic localization of functions can be used to keep in the workspace only those functions directly required by functions currently in the *SI*. This gives *SPM* a dynamic overlay mechanism.

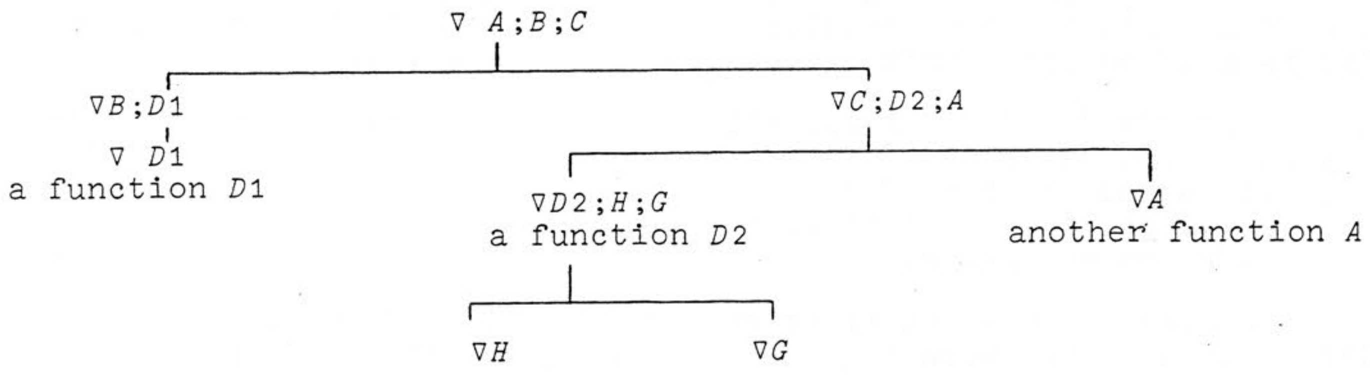
An *SPM* system can be thought of as a tree structure, with functions as "nodes" of the tree: sons of a given function are functions local to it. An *SPM* system thus has a static structure much like an ALGOL-60 or a PL/1 program.

The functions held in the workspace are:

(a) - functions on the path from the root of the tree to the function currently being executed and

(b) - those functions directly local to functions in (a). These form the working set for a particular function.

SPM Localization Structure:



Suppose *A* calls *B*, and *B* calls *D*1. When *A* is called, *B* and *C* will be defined in the workspace, local to *A*. When *B* is called, *D*1 will be defined in the workspace, local to *B*. When *D*1 is being executed, the following functions will be in the workspace:

A - root function, workspace global
B and *C* - local to *A*
*D*1 - local to *B*.

Note that dynamic localization allows the existence of several functions with the same name. There are two functions called *A* in the tree above: the first is at the root of the tree and could be the master program, while the second is local to a function local to the first *A*. The second *A* might be quite different. This feature of dynamic localization allows one to merge subsystems written by several programmers in which the same name may have been used for different functions.

MAINT - This facility is used to build and maintain a system of functions on files. Two basic files are used: a function text file and a directory file for the text file. Function texts are stored in 2 ⎕*FD* form (as matrices) including comments. This form is convenient for line by line editing and for insertion and deletion of comments, although it may require more storage space than 1 ⎕*FD* format. The directory organizes and interrelates the individual functions. For programming convenience the system is divided into subsystems of up to 100 functions. A subsystem may be self-contained, or it may use functions from other subsystems. In this respect a subsystem is a more flexible structure than a workspace which cannot dynamically share programs with other workspaces. The maintenance program recognizes "system commands" and "subsystem commands" (*SAVE*, *LOAD*, *ERASE*, etc.), suggesting further analogies with workspaces.

A subsystem is, however, supplied with further structural information in the form of two graphs (represented as boolean matrices):

Functions on files can be used in a variety of systems at different levels of complexity. Simple hybrid systems offer a good way to beat *WS FULL* problems at minimum cost, while complete systems like *SPM* can help to maintain, document and expand very large, complex systems.

Thank you for your comments on the timing functions. A common problem was a failure to ensure that the number of repetitions (N) was large enough.

i.e. if ($N \times$ Result) < 1000 - the results are unreliable.

Questions, comments and contributions are welcome and should be addressed to:

Clement Kent, (mailbox code *KENT*)
I.P. Sharp Associates Limited,
Suite 1400, 145 King St. W., Toronto.